Teaching Architectural Approach to Quality Software Development through Problem-Based Learning

Kam Hou Vat
University of Macau, Macau, China

fstkhv@umac.mo

Abstract

This paper describes the initiative to incorporate the practice of quality software development (QSD) into our undergraduate curriculum concerning the engineering of software. We discuss how the constructivist's method of problem-based learning (PBL) helps develop this QSD practice into our students' daily learning. This paper expounds the idea of an architectural approach to building software solutions, which is supported by the industry's emerging consensus that architectural components provide the kind of building blocks we need for developing today's complex systems. Particularly, the technology of component-based development asks of us the required portions of productivity, quality, and rapid construction of software artifacts. Consequently, our pedagogic approach to QSD focuses on designing and building a sensible architecture characterized by objects of different services, which represent the cohesive collections of related functionality, accessed through some consistent interfaces that encapsulate the implementation. The paper outlines an QSD approach in terms of state-of-the-practice development processes modified for educational scenarios, through which our students could learn to acquire their collaborative software engineering experience in the current practice of architected application development. The paper concludes by discussing the criteria used to evaluate the working of the learning scenario and some lessons learned involved in incorporating PBL learning scenarios suitable for QSD.

Keywords: Quality Software Development, Component-Based Development, Problem-Based Learning

Introduction

The term 'software engineering' was first coined as the theme of the NATO-sponsored meetings in 1968 and 1969 (Software Engineering 1969) to assess the state and prospects of software production (Shaw 1990). Currently it has come to imply not only a set of current practices for software design and development. But it also represents a growing discipline whose engineering characteristics are being increasingly materialized in the form of recognized skills and knowledge needed for the certification, registration, and licensing of software engineers (McConnell and Tripp 1999; Pour, Griss and Lutz 2000; Speed 1999). Though the software maturation process (Bourque et al. 1999; Lethbridge 2000; Meyer 2001; Parnas 1999; Wasserman 1996) has witnessed consolidation of existing and emergent development methods and tools, as well as a widely accepted consensus on its intellectual content by both academic and professional bodies, we are often confronted with the situation (Dawson and Newsham 1997) that most Software Engineering (SE) graduates today begin their careers, lacking an appreciation of real-world conditions. As academics, the haunting question is this: How do we cultivate future software professionals, who could become eligible for creating quality software solutions to practical problems, starting from their university education of software engineering? This paper is an educational response to this imminent

situation to devise suitable quality software development (QSD) experience for our undergraduate SE students. In the following discussion, we first introduce our pedagogy of problem-based learning (PBL), followed by a briefing on the software development scenario for modern enterprises. Then we expound an architectural approach to building software solutions, including the team-based activities for managing software requirements, which is to be learned and practiced by our students through problem-based learning. Finally, we discuss the evaluation criteria used for our scenario-based design and point to some lessons learned from incorporating PBL for QSD in our experimentation.

The PBL Model of Investigation

The notion of PBL is based on the premise that students learn more effectively when they are presented with a problem to solve rather than just being given instruction (Greening 2000; Ryan 1993). Pedagogically, students have to identify and search for the knowledge they need to approach the problem. When applied to the course setting, PBL could be decomposed into several stages of activities (Albanese and Mitchell 1993; Barrows 1985; Perkins 1992; Savery and Duffy 1995), which help develop in students, self-directed learning and problem-solving skills while they interact, discuss and share relevant knowledge and experience.

- *Problem Analysis Stage.* Students, divided into small groups and assigned a facilitator, are respectively presented a problem without any instruction given. They generate ideas about possible solutions to the problem based on what they already know. They then define what they need to know by identifying the key learning issues and formulate an action plan to tackle the problem.

- *Information Gathering Stage.* A period of self-directed learning follows. Students are responsible for searching for relevant information. They are largely engaged in just-in-time learning as they are seeking for information when their need to know is greatest.

- *Synthesis Stage.* After a specified period of time, students reconvene and reassess the problem based on their newly acquired knowledge. They become their own experts to teach one another in the group; they use their learning to re-examine the problem. In the process, they are constructing knowledge by anchoring their new findings on their existing knowledge base.

- *Abstraction Stage.* Once the students feel that the problem task has been successfully completed, they discuss the problem in relation to similar and dissimilar problems in order to form generalizations.

- *Reflection Stage.* At this stage, students review their problem-solving process through conducting a self- or peer-evaluation. This phase is meant to help students' meta-cognitive ability as they discuss the process and reflect on their newly acquired knowledge.

The Teamwork Foundation for QSD

Essentially, PBL revolves around a focal problem, group work, feedback, class discussion, skill development and final reporting. The instructor's role is to organize and pilot this cycle of activity, guiding, probing and supporting students' initiatives along the way so as to empower them to be responsible in their own learning. Nevertheless, it is important that PBL students must be taught how to work in teams and positively experience the team process because the team skills they acquire are applicable throughout their future careers. Overall, our teaming concerns cover such particulars as team structure, formation, activities, and assessment in support of our pedagogical purpose (Brown and Dobbie 1998; Wills 1998).

- *Team Structure*. The PBL approach requires each team composed of 3-5 students, to be assigned a supervisor (instructor) and a client. The client's role is to clarify the project, and to resolve ambiguities as they arise, whereas the supervisor's is to guide, motivate and provide feedback to the team. Also, one of the team members is designated the team leader for the duration of the project, whose role is to coordinate

the team activities, and to ensure effective team communications. The leader also has to interface with the supervisor, arrange meetings with clients when necessary, and facilitate meeting through setting agendas, taking minutes, and allocating tasks. Each team member has to help set the team goals, accomplish tasks assigned, meet deadlines, attend team meetings and take a turn editing a document to be submitted at the end of each major stage of project development.

- *Team Activities*. PBL students are made aware of the difficulties in teamwork throughout the project period. These include setting realistic project goals, carefully allocating tasks to team members, managing time, and communicating and managing shared group documents. Teams have regular meetings to which they invite their supervisor, and in which they organize themselves to manage the project. Students are often reminded of setting appropriate agendas before meeting, assigning enough time to the agenda items during meeting, restating the decisions made at the meeting, and converting decisions into action items after meeting. They are also advised on clearly separating the social and work aspects in meetings, and assessing each meeting for doing it better next time. Moreover, it is suggested that teams plan their project around major deadlines of individuals in the team thereby acknowledging the other commitments team members may involve.

- *Team Assessment*. Deadlines represent the milestones set down for the PBL students to submit project documents and to receive evaluation. Each member is assessed by the project supervisor and the team peers. The supervisor's evaluation is based on what each team member adds to the meetings and what the instructor perceives each member's contributions to the team to be. The peers' evaluation is based on a confidential rating sheet, to be completed by each team member at the end of each major phase of the project. This rating sheet should include each team member's contribution for that phase with explanatory comments. And the overall project assessment is made up of the group grade and the individual grade. The former is the same for each group member and is based on the quality of the documents produced and the product developed. The individual component is based on the quality of the student's contribution to the documents and the product, their participation in group-meetings, their commitment to the team process, and their professional attitude developed.

- *Team Formation*. In forming teams, the PBL approach invites students to complete a questionnaire asking for details such as courses taken, programming languages and tools familiar, possible experience in teamwork, willingness to be a team leader, and some information on their current study and work schedule. Such information is then used to group students in individual teams with continuous guidance and counseling to resolve possible conflicts.

The Curriculum Arrangement for QSD

We agree with Meyer and others (Boehm and Basili 2000; Meyer 2001; Wasserman 1996) that much of the body of software engineering knowledge consists of a set of recurring concepts that software professionals learn through trial, error, and skillful mentoring. Typical examples include the idea of abstraction, the distinction between specification and implementation, the practice of iterative development, and the initiative to design for change. In order to enable our students' learn and practice of such important know-how, most of our current software engineering undergraduate programs today, are designed to include a core curriculum with the practice of system development, and the option of specialization. The core curriculum comprises a set of required courses that provide the basic knowledge and skills, emphasizing design, analysis and even the management of software systems. There is often a semester-long or yearlong project course, which gives students hands-on experience in developing software system through applying the materials learned in class. Besides, students are often allowed to develop a specialization in specific areas of software engineering, through making a selection of elective courses in their senior years. In the case of the author's affiliated university, we are proposing that undergraduate SE students are given opportunities to involve themselves in annual project assignments. These projects, preferably starting from

the junior year and each spanning two regular semesters (fall and spring), should require students to plan and implement a significant software system for a real client (be it from inside or outside the university environment). In each annual project, students have to work as a team under the guidance of faculty advisors, to analyze a problem, plan the software development effort, execute the solution, and evaluate their work. Pedagogically, the second annual project often targeted as the senior graduating project has been in place for years (though it has been just one-semester long for quite some years), but the first one which serves as the juniors' warm-up exercise to acquire real-world experience, is often missing. To compensate for this important junior year experience, the author has been revising his junior courses such as human-computer interaction (Vat 2000, 2001) to make room for students' practical PBL experience. It is expected that we could soon properly install the two yearlong projects for the students to learn and accumulate their software development expertise for their future benefits.

The Learning Scenario for QSD

It is understood that collaborative project work is recognized as having many educational and social benefits, in particular providing students with opportunities for active learning. However, teaching, directing and managing group-based project work is not an easy process. This is because projects are often: *expensive* demanding considerable supervision and technical resources; and *complex* combining design, human communication, human-computer interaction, and technology to satisfy objectives ranging from consolidation of technical skills through provoking insight into organizational practice, teamwork and professional issues, to inculcating academic discipline and presentation skills. Fortunately, PBL as a process-based instructional method does help preparing our students to get started with group project work to initiate their immediate journey as future software professionals. Our learning scenario for QSD, based on real-world findings, is designed to arouse our students' attention to the following situation of concern:

> *Most businesses today are undergoing a period of rapid change, driven by trends such as business-process improvement and downsizing (Cook 1996; Umar 1997). In the past, the order of the day has been to re-organize the technology each time a business changes. Yet, the re-oriented consensus is to facilitate software solutions (technology) that adapt as the business adapts. This support for increasingly adaptive businesses is currently achieved through the reuse of business components (Eeles 2000), which are executable units of code that provide physical black-box encapsulation of related business services, accessed through a consistent, published interface that includes an interaction standard with other components. These business components support a process-based view of the business as it changes. Consequently, it is important to derive the business process models in order to provide a secure business foundation from which to develop component-based solutions, which are necessarily traceable back to originating business requirements.*
>
> *On the other hand, one of the main enabling technologies for component reuse is the standardization of distributed-object middleware (Berstein 1996), through which components can be moved around at execution time and deployed in a way that optimizes the technology in order to deliver the most business benefit. These advances in component technology have resulted in the movement toward separation of software applications from the increasingly heterogeneous technology platforms on which the services are deployed (Anderson and Dyson 2000; Cook 2000). It provides the potential for an application to be physically distributed so that it services the needs of the business and not the technology. We call this the service-based view of software construction, where components provide a method of packaging related services into pre-fabricated pieces of software from which solutions can be constructed. This service-based approach is also applicable in the area of legacy software where most development is about enhancing existing systems, providing new front-ends to established back-ends, capitalizing on existing relational technology for data storage, and building interfaces to existing packages. It allows organizations to wrap the existing services into*

new offerings or products, so as to reuse their investments in existing packages, databases, and legacy systems within the context of component technology.

The overall picture confronting today's enterprises could be characterized as this (Allen and Frost 1998; Cook 1996; Cook 2000): at the core is the installed base of existing information technology (IT) systems, which includes the legacy data and business logic. Around the edge are increasingly pro-active customers, to which the enterprise must offer an increasing quality of service through existing and new channels. In between, the enterprise is re-engineering its business processes, with a focus on knowing its customers better, and offering continuous improvement of its products and services. From an IT perspective, the legacy systems become surrounded by a matrix of go-between componentry providing services to support the changing business, with increased flexibility and reduced development times as compared with legacy systems. This is often a challenge requiring skilled and thorough design, taking into account such attributes as reliability, efficiency, usability, maintainability, testability, portability and the most essential reusability. Nonetheless, it is worthy to point out that modern businesses today require applications that confer early user benefits at minimum cost, leveraging existing legacy systems where possible but not at the cost of maintainability, flexibility and reusability.

Meanwhile, a common reaction to the pressure of immediate business needs is to virtually abandon planning and control in the name of producing fast results (Gartner Group 1995). However, incremental releases that are developed in isolation solely to meet tight deadlines will eventually result in fragmented systems that lack consistency and fail to provide integrated support. Worse still, this presents a problem that grows out of control exponentially with the number of increments delivered. We believe that a key requirement of an incremental approach is to base increments on a sound architecture that enables components to be plugged in as service providers to the increments. Besides, this should be an architecture for model building, which supports such goals as management of scale and complexity, interoperability, and adaptability. In large organizations with complex business processes, there is a need to manage scale and complexity in software development in such a way that the resulting software structure mirrors business needs as closely as possible (Gartner Group 1996). This requires the architecture to establish the definitions, rules, and relationships that will form the infrastructure of models from business process to code. More, this architecture should support the idea of software evolution among a mix of legacy systems and databases, off-the-shelf packages and newer object-oriented applications.

This is the backdrop behind which most of the real-world QSD projects have been conducted, despite the fact that it has not been consciously made known to our software graduates (Favela and Pena-Mora 2001). The important issues often boil down to the following "how-to's" (Allen and Frost 1998; Braude 2001). These include: support increasingly adaptive businesses; capitalize on the rapid advances in component technology; deal with legacy systems; plan and build for reuse; prepare for quality issues; and retain a pragmatic focus in the face of increasing complexity. Collectively, they represent the drivers of change, worthy enough to secure a place in our SE students' curriculum of study.

Architectural Approaches for QSD

The architectural approach designed for our learning scenario is based on fundamental principles (Hartley, Hruschka, and Pirbhai 2000) that we believe are applicable to system development regardless of the underlying techniques used. Briefly, we believe each system is a component of one or more larger systems. The larger systems are the context or environment in which the component system must work. Likewise, systems comprise a layered set of subsystems below the layer with which we happen to be dealing, and a layered set of super-systems above that layer. This layered structure can be exploited both in representing systems and in defining the system development process. Indeed, most systems are members of multiple

layered sets. The particular set(s) chosen to represent a system are determined by the viewpoint(s) that are important for the particular system. Also, every system has a set of essential requirements, which meet the needs of the context or environment, without imposing any specific implementation, and a set of physical requirements, which reflect the architectural and design decisions made to satisfy the essential requirements. To carry out the essential requirements, and thereby to meet the needs of the environment, systems receive as inputs, produce as output, and process internally the necessary information. To succeed in the development of complex systems, all system artifacts invoked by these principles must be represented separately, but with their relationships and interactions also represented. To help our PBL students embark on their journey of system development, we have selected some recurring concepts and state-of-the-art architectural thinking for them to learn through trial, error and mentoring:

Managing Complexity through Models

A model is an abstraction highlighting some aspects of real-world systems in order to depict those aspects more clearly. A model has an objective (the question we want it to answer) and a viewpoint (the point of view of one or more stakeholders: users and developers). Abstract models reduce the complexity of the real world to digestible chunks that are simpler to understand. Different types of models answer different types of questions about the system they represent. If we decide to build more than one model of a given system to investigate different aspects, then we should somehow organize these models according to their relationships to one another and to the system. That is why we often need a framework to accommodate different models.

Separating Concerns of What and How

Every system has a specification comprising two important parts: system requirements and system architecture. Both of these parts contain models. The 'system requirements' model is a technology-independent model of the problem the system is to solve. It represents the 'what'. The 'system architecture' model is a technology-dependent model of the solution to the problem. It represents the 'how'. Typically, these two models are created for the entire system and for every sub-system down to the lowest level in the system hierarchy. And it is important to separate the 'what' and the 'how' for the following reasons: It is often very useful to understand a problem independently of any particular solution. Any given problem has many possible solutions. Selection of a particular solution is a trade-off process; we often need to make numerous different trade-offs while keeping the problem statement unchanged. The separation supports the generally recognized principle of separation of concerns, which means dealing with only one part of the system's complexity at a time. The 'requirements' model only has to cope with essential problems; the 'architecture' model has to cope with many constraints imposed by technology and organization. This separation of the 'what' and the 'how' gives us the power to re-implement the 'what' using new technology, but it also gives us the convenience of reusability – not just for software or hardware, but for requirements as well. This is particularly important because requirements are relatively more stable over longer periods of time than technology.

Developing Layered Models

In any discussion of systems, models of systems, or the process of building systems, the term 'layer' plays an important role. We can identify several basic attributes and relationships that keep recurring for our clarification. The typical attributes include the following: The layers, once they are identified, form a structure that can be interpreted and developed in any sequence and independently of one another. The number of elements per layer typically increases downward, giving the whole structure a pyramidal shape. These elements forming the layered structure can be considered a set, either of activities or of entities. Such elements may be carried out or used in some prescribed sequence, concurrently, or in any combina-

tion of sequence and concurrency. Also, elements in the layers usually communicate and cooperate up, down, and sideways within and between layers. Moreover, each layer includes, deals with, or is associated with some requirements, some architecture or design, some construction or implementation, and some integration and testing. Meanwhile, there are many types of relationships in layered models, but four of them are of special interest in our learning scenario: aggregation/decomposition, abstraction/detailing, super-type/subtype, and controlling/controlled.

- *Aggregation/Decomposition Relationships.* Through this relationship, elements in the higher layers actually consist of the elements in the lower layers, or conversely, elements in the lower layers are decompositions of those in the higher layers. This structure is also known as a whole/part structure or a container/content structure: namely, a given layer provides the container for the layer below, which is the content of the layer above. In practice, an aggregate involves more than just collecting sub-elements into a set. The sub-elements must also interface with one another, requiring linkages between them that may not be evident when they are considered separately. When applied to software, we can imagine an architecture module at the highest software layer; major subprograms it contains are modules in the next layer down; sub-subprograms or subroutines form a further layer.

- *Abstraction/Detailing relationships*. Through this relationship, the higher layers are simply more abstract expressions of the lower layers, or conversely, the lower layers are more detailed expressions of the higher layers. It is important to notice this. An abstract requirement statement does not contain the more detailed requirements statements that describe it. If we assemble a set of detailed requirements, we merely have a collection of detailed requirements – the abstract and detailed requirements exist independently of each other, with an abstraction/detailing relationship between them.

- *Super-type/Subtype Relationships*. Through this relationship, an element in the higher layer – the super-type – includes all of the features that are common to its associated elements in the lower layer – its subtypes. These features are attributes that are inherited by the elements on the lower layer. Starting from the lower level, super-types are formed for sets of elements that share common attributes. Super-type/subtype models are important in object-orientation. This relationship is the foundation for inheritance. Moreover, object orientation has taken this relationship and extended it to more complex forms of inheritance than just attribute inheritance: The lower layer may also inherit functions and the behavior of the super-types. With the super-type/subtype relationship, it is important that the super-type contains all the commonalities of the subtypes. The main use of this relationship is to discover commonalities and to describe them only once, thus reducing redundancy. The structure then allows the lower layers to inherit whatever commonalities have been discovered. We can see that this relationship is a subtype of the abstraction/detailing relationship. A super-type is an abstraction of its subtypes, and the subtypes are detailed instances of the super-type. Other names used for this relationship include generalization/specialization, class hierarchies, inheritance structures, and ‘is-a’ hierarchies.

- *Controlling/Controlled relationships*. Through this relationship the upper layers control elements of the lower layers. Other terms used for this relationship are control hierarchy, or the is-boss-of (is-supervised-by) relationship. Sometimes, we simply say that the higher element uses the lower elements. The higher layer must have knowledge of the lower layer but the lower layer – that is, the one being used – does not necessarily have to know anything about the boss. In terms of client/server models, the client is the boss that delegates work to the server; the server provides certain services that are performed whenever a client asks for them.

- *Layered Models Summary*. The four types of relationships in layered models are distinctly different from one another; they all serve distinct and important roles in system development; and they can be integrated smoothly, where appropriate, with other models, including object-oriented models. Typically, layered models based on the four types of relationships can be used simultaneously to represent different aspects of one system. For example, the required functional capabilities of a system can be captured by a

process model, which is based on the 'abstraction/detailing' relationship. The required behavioral capabilities are captured by a control model, which is based on a 'controlling/controlled' relationship. The information structures in the system are captured in an entity-relationship model based on the 'supertype/subtype' relationship. Also the physical structure can be captured by the architecture model, which is based on an 'aggregation/decomposition' relationship. In sum, the layered models allow us to represent different views of the system separately, but when done as part of the requirements and architecture models, the links between these views are carefully maintained.

Investigating Typical Architectural Approaches

There are different schools of architectural approaches to building software solutions. Each school is relatively isolated from the others, sharing concepts and principles, but employing widely different terminologies. We have singled out three alternative approaches for our students' investigation: the Zachman Framework (Inmon, Zachman and Geiger 1997; Zachman 1987), the Reference Model for Open Distributed Processing (RM-ODP) (ISO 1996), and the 4+1 View Model (Booch, Rumbaugh and Jacobson 1999). Here is a brief tour of the major schools of architectural thinking.

- *Zachman Framework*. This approach, originated from IBM research and practice, is a traditional architecture approach, which predated the popularity of object orientation. In the Zachman Framework, there are six information system viewpoints as well as five levels of design abstraction. The Framework is a reference model identifying a total of 30 specifications, in the form of a matrix of architectural descriptions organized in terms of the intersection of two paradigms: journalistic questions (who, what, when, why, where, and how) and information system construction (planner, owner, builder, designer, subcontractor). System architects choose from among these 30 architectural viewpoints to specify the system architecture. In practice, real-world project is hardly capable of creating these 30 or more detailed plans and keeping them all in synchronization. Beyond the Zachman Framework, object-oriented architects have discovered additional needs for defining computational architecture and other viewpoints which are not obvious applications of the Zachman principles. Recently completed is the ISO reference model for open distributed processing called RM-ODP.

- *ISO RM-ODP*. This model belongs to a category of ISO standards called open distributed processing (ODP). ODP is an outgrowth of earlier work by ISO in open systems interoperability, involving object-oriented and distributed computing paradigms that has achieved stability, multi-industry acceptance, and formal standardization. ODP defines a five-viewpoint reference model (enterprise, information, computational, engineering, and technology), and a comprehensive set of terminology, a conformance approach, and viewpoint correspondence rules for traceability. Among the various architectural approaches, RM-ODP defines what information system architecture means, and is a model representative of mature software architecture practice today. The five essential viewpoints for modeling systems architecture can be summarized as follows. An enterprise viewpoint contains models of business objects and polices. It assures that business needs are satisfied through the architecture and provides a description, which enables validation of these assertions with the end users. An information viewpoint includes the definition of information schemas as objects (static, invariant and dynamic). The perspective is similar to the design information generated by a database modeler. It is a logical representation of the data and processes on data in the information system. A computational viewpoint includes definitions of large-grained object encapsulations, including subsystem interfaces and their behaviors. It partitions the system into software components, which are capable of supporting distribution. It defines the boundaries between software elements in the information system. Generally, these boundaries are the architectural controls that assure that the system structure will embody the qualities of adaptability in management of complexity that are appropriate to meet changing business needs and incorporate the evolving commercial technology. Finally, an engineering viewpoint exposes the distributed nature of the information system. It defines the mappings between the engineering objects and other architected objects to specific standards and technologies

including product selections. The distribution transparencies supported by infrastructure are declared explicitly. The allocation of objects onto processing nodes is also specified.

- *The 4+1 View Model*. This is a viewpoint-based architectural approach supported by object-oriented tools such as Rational Rose. The viewpoints include use-case view, logical view, process view, implementation view, and deployment view. The use-case view models enterprise objects through a set of scenarios. The logical view includes object models of packages, classes, and relationships. The process view represents control flows and their inter-communications. The implementation view defines the modular structure of the software. The deployment view identifies the allocation of software onto hardware. An architecture defined as a 4+1 View Model covers aspects of all five RM-ODP viewpoints.

Addressing Component-Based Development

Our study of component-based development (CBD), based on (Allen and Frost 1998) model, is evolutionary in nature. We aim to harness a service-based method with effective object-oriented modeling to capitalize on the increasing power of the fast-developing component technology. The idea is to provide an overall design philosophy for realizing the vision of service-based reuse of components (Allen and Frost 1998; Anderson and Dyson 2000; Cook 2000). We call this philosophy the *service-based architecture* for CBD, which employs the concept of *service packages*, to facilitate a business-oriented modeling process. A service package provides a set of services belonging to a single service category. Each service from a service package is realized by an individual component, which is also a container of different objects. This provides a business-oriented basis for modeling deployment of components using *services packages*, which are implementation packages of objects, providing services through their interfaces. That way, components provide a means of packaging related objects together into pre-fabricated pieces of software. And service packages provide a mechanism for grouping those objects into units (in the form of components) that are cohesive to the needs of a particular set of services from which business solutions can be constructed. It is important to notice that the promise of component-based development is that software solutions can be composed from reusable components, in analogous fashion to hardware (Cox 1986; Eeles 2000; Repenning et al. 2001). Nevertheless, the service packages must be modeled in a way that makes the resulting components useful building blocks, simple to activate and inexpensive to administer. The level of granularity of a component can vary from large and complex to small and simple. In practice, large components have the greatest potential for reuse but are often not cohesive and may be difficult to assemble into solutions with other components. Small components are usually more cohesive but often need to be coupled with many other components to achieve significant reuse, resulting in excessive inter-component coupling. Clearly, settling on a good and useful level of granularity is a trade-off between these two extremes.

Adapting a Process for Architected Applications Development

The term “process” as used in our pedagogic context for architected applications development (AAD), carries the connotation of process models designed to view the real world from the viewpoint of architectural software development. Thus, the process to be described is an abstract description of the software development activities within the service-based architecture (Stapeton 1997). We are interested in a two-tier process to achieve an AAD methodology: the solution process, and the component process. The former is aimed at development of solutions, typically in terms of user services, to maximize reuse of existing services and provide early user value. The latter is aimed at developing components that provide commonly used business and/or data services across different departmental systems or for use by third parties. It is important to notice that we often need to use elements of both processes adapted to our own specific needs. Typically, the key driver of the solution process is a set of specific requirements to meet the needs of a business process. Various models are produced throughout the process, which evolve in

detail as the process unfolds. We continually seek opportunities to extend and refine existing generic models. Such models are often selected on a use-case by use-case basis for incremental development of user services. On the other hand, the generic business requirements that drive the component process may come from the need to reuse existing legacy assets, and the feedback from solution projects. An important part of the component process is to evolve the models so that they can be specialized and refined by solution builders to form an evolving set of components. The use of a process by a software development team should assist project management in numerous tasks. These include: identification and partitioning of work, identification of progress achieved, planning the staff resource profile, planning the requirement for physical resources, and provision of cost and time scale estimates for the work yet to be performed. From a technical viewpoint, a process should assist in such areas as identification of preconditions required before each activity is started, specification of the products and deliverables required from each activity, techniques that may be used during each activity, and experience gained from earlier work. Clearly, building and refining generic models is an important aspect of CBD where we want to leverage model reuse more than code reuse. Service packages provide a means of structuring a project in terms of architectural context and allow us to build on and capitalize on the very best work of others. A service package can be effectively employed in a component process to contain a generic model, which can be refined and extended to meet the specific needs of a solution process. The model solution space evolves to contain more detail as a project moves through the iterations of the process. Eventually, portions of the model are mature enough to be transformed into code. The tested code represents the model at its most detailed level of abstraction. As for deliverables, they are simply views of a maturing model. In practice, the service-based process for AAD is very much an adaptive process that can be tuned and customized to specific organizational needs. Checkpoints can also be built into the process to help evolve it. This includes documenting the lessons learned so that others can avoid making the same mistakes.

Team Skills Development for Managing Software Requirements

To partially address the requirements challenge in architectural software development, we have selected the following team-based activities (DeMarco 1982; Jacobson et al. 1992; Leffingwell and Widrig 2000) to be experienced by our PBL students within their annual project duration.

- *Analyzing the Problem*. This includes a set of skills to understand the problem to be solved before application development begins. It is the process of understanding real-world problems and user needs and proposing solutions to meet those needs. We consider a problem as the difference between things as perceived and things as derived (Gause and Weinberg 1989). Accordingly, if the user perceives something as a problem, it is a real problem, and it is worthy of addressing. Typical techniques include gaining agreement on the problem definition; understanding the root causes to induce the problem, and identifying the stakeholders and the users, with the former being anyone who could be materially affected by the implementation of the new application.

- *Understanding User Needs.* This introduces a variety of techniques to elicit requirements from the system users and the stakeholders. Software teams are rarely given effective requirements specifications for the systems they are going to build. Often they have to go out and get the information they need to be successful. Typical methods include interviewing and questionnaires, requirements workshop, brainstorming and idea reduction, storyboarding, use cases derivation, role playing, and prototyping. Each represents a proactive means of pushing knowledge of user needs forward and thereby converting fuzzy requirements to those that are better known.

- *Defining the System.* This describes the initial process by which the team converts an understanding of the problem and the users' needs to the initial definition of a system or application that will address those needs. Our PBL teams should learn that complex systems require comprehensive strategies to organize information for requirements. This information could be expressed in terms of a hierarchy, starting

with user needs, transitioning through feature sets, then into the more detailed software requirements. The latter could be expressed in use cases or traditional forms of requirements documents, say, the vision document defining at a high level of abstraction, both the problem space and the solution space.

- *Managing the Project Scope.* This reminds our teams that they should be aware not to initiate projects with too large a scope to be accomplished. Project scope is presented as a combination of the functionality to be delivered to meet users' needs, the resources available for the project, and the time allowed in which to achieve the implementation. The purpose of scope management is to establish a high-level requirements-baseline for the project. The team has to establish the rough level of effort required for each feature of the baseline, including risk estimation on whether implementing it will cause an adverse impact on the schedule. Also, each team has to actively engage its customers in helping solve the scope management problem to ensure both the quality and the timeliness of the software outcomes.

The Criteria for PBL Evaluation

Throughout our project duration, we have born in mind that our instructional method should be evaluated in part by its ability to explain practice. The following explicit criteria (Greening 1998; Ryan 1993; Savery and Duffy 1995) have been defined, in order to later judge the learning outcome with respect to the process of problem diagnosis, action intervention, and reflective learning.

- *Learning is an active and engaged process.* Instead of being told what to do or how to solve problems, students within a PBL atmosphere are to generate their own learning issues. It is expected that a sense of ownership should be born leading to greater cognitive engagement. Students are actively engaged in working at tasks situated in an authentic setting which should lead to greater ability in transfer to other real-world contexts.

- *Learning is a process of knowledge construction.* PBL purports that learners construct their own knowledge. The constructivist epistemology states that the known is internal to the knower and is subjectively constructed based on individual responses to experience. Thus, in order to harness the reality of learning, we need to consider the opportunity to find knowledge for oneself, contrast one's understanding of that knowledge with others' understanding, and refine or re-structure knowledge as more relevant experience is gained.

- *Learners function at a meta-cognitive level.* Constructivist learning focuses on initiative thinking activities rather than working on the 'right answer the teacher wants'. Students generate their own strategies for problem formulation and possible solutions. The instructor's role is that of a facilitator, a guide or a coach, probing students' thinking, monitoring their activities, and generally keeping the process moving. Thus, PBL should promote meta-cognition through encouraging students to reflect upon the problem-solving process. It is believed that reflection on recent experiences is an effective method of learning.

- *Learning involves social negotiation.* We accept the constructivist perspective that knowledge is socially negotiated. The quality or depth of one's understanding can only be determined in a social environment where we can see if our understanding can accommodate the issues and views of others and to see if there are points of view which we could usefully incorporate into our understanding. The importance of a learning community where ideas are discussed and understanding enriched is critical to the development of our students into self-directed work teams of software professionals.

Lessons Learned

It is experienced that the conventional approach to education remains the instructivist one, in which knowledge is perceived to flow from experts to novices. This transmissive view of learning is most evident in the emphasis on lectures, in the use of textbooks to prescribe reading, and in the nature of tutorials

and assessment methods. It assumes that the process of good teaching is one of simplification of the truth in order to reduce student confusion. Yet, this simplification could deny students the opportunity to apply their learning to dynamic situations such as quality software development through team-based collaboration. We question the transferability of the instructivist learning and ask how much of that which is assigned to academic learning ever gets applied to actual scenarios, when there is such a rapid surge in knowledge commonly associated with the birth of the "Information Age." This is a transference problem. Actually, the content product of learning is assuming a less important role relative to the process of learning as the life of information content shortens and the need for continual learning increases. In designing the learning scenario for QSD to be injected into our project courses for software engineering, we have tried to reorient towards a meaningful direction by reducing the obsession with knowledge reproduction. And PBL represents one such relief from the constructivist pedagogy. Greening (Greening 2000) describes it as a vehicle for encouraging student ownership of the learning activities. There is an emphasis on contextualization of the learning scenario, providing a basis for later transference, and learning is accompanied by reflection as an important meta-cognitive exercise; for example, assessing whether a project should be approached by a solution process or a component process. Also, the implementation of PBL is done via group-based work, reflecting the constructivist focus on the value of negotiated meaning (Perkins 1992). More importantly, it is unconfined by discipline boundaries, encouraging an integrative approach to learning, which is based on requirements of the problem as perceived by the learners themselves. In conclusion, this paper has described our approach of quality software development through the various examples of architectural thinking, including CBD and the corresponding solution and component processes for AAD. We have also explained how to incorporate this QSD-based practice into our curriculum through the PBL pedagogy.

References

Albanese, M., and Mitchell, S. (1993), "Problem-Based Learning: A Review of Literature on Its Outcomes and Implementation Issues," *Academic Medicine*, Vol. 68, No. 1, pp. 52-81.

Allen, P., and Frost, S. (1998). *Component-Based Development for Enterprise Systems: Applying the SELECT Perspective*, Cambridge University Press.

Anderson, B., and Dyson, P. (2000), "Reuse Requires Architecture," In L. Barroca, J. Hall, and P. Hall (Eds.), *Software Architectures: Advances and Applications*, Springer-Verlag, Berlin Heidelberg, pp. 87-99.

Barrows, H.(1985). *How to Design a Problem-Based Curriculum for the Pre-Clinical Years*. New York: Springer.

Berstein, P. (1996), "Middleware: A Model for Distributed System Services," *Comm. ACM*, Vol. 39, No. 2, Feb., pp. 86-98.

Boehm, B., and Basili, V.R. (2000), "Gaining Intellectual Control of Software Development," *IEEE Computer*, May, pp.27-33.

[Booch, G., Rumbaugh, J., and Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison Wesley, Reading, Mass.

[Bourque, P., Dupuis, R., Abran, A., Moore, J.W., and Tripp, L. (1999), "The Guide to the Software Engineering Body of Knowledge," *IEEE Software*, Nov.-Dec., pp. 35-44.

Braude, E.J. (2001). *Software Engineering: An Object-Oriented Perspective*. John Wiley & Sons, Inc.

Brown, J., and Dobbie, G. (1998), "Software Engineers Aren't Born in Teams: Supporting Team Processes in Software Engineering Project Courses," In Proceedings of IEEE International Conference on Software Engineering: Education & Practice, Dunedin, New Zealand, Jan. 26-29.

Cook, M.A. (1996). *Building Enterprise Information Architectures: Reengineering Information Systems*, Prentice Hall PTR.

Cook, S. (2000), "Architectural Standards, Processes and Patterns for Enterprise Systems," In L. Barroca, J. Hall, and P. Hall (Eds.), *Software Architectures: Advances and Applications*, Springer-Verlag, Berlin Heidelberg, pp. 179-190.

Cox, B. (1986). *Object-Oriented Programming: An Evolutionary Approach*. Reading, MA: Addison-Wesley.

Dawson, R., and Newsham, R. (1997), "Introducing Software Engineers to the Real World," *IEEE Software*, Nov., pp.37-43.

DeMarco, T. (1982). *Controlling Software Projects*. Eaglewood Cliffs, NJ: Yourdon Press.

Eeles, P. (2000), "Business Component Development," In L. Barroca, J. Hall, and P. Hall (Eds.), *Software Architectures: Advances and Applications*, Springer-Verlag, Berlin Heidelberg, pp. 27-59.

Favela, J., and Pena-Mora, F. (2001), "An Experience in Collaborative Software Engineering Education," *IEEE Software*, Vol. 18, No. 2, March/April, pp. 47-53.

Gartner Group, (1996). Best Practices in Application Development Project Management, Part 2, SPA-650-1293. ADM Research Note, March 20.

Gartner Group, (1995), "Rapid Application Development, Part 2: Organizing for Success," Inside Gartner Group This Week, June 7.

Gause, D., and Weinberg, G. (1989). *Exploring Requirements: Quality before Design*. Dorset House Publishing.

Greening, T. (2000), "Emerging Constructivist Forces in Computer Science Education: Shaping a New Future?" In T. Greening (Ed.), *Computer Science Education in the 21st Century*, Springer, pp. 47-80.

Greening, T. (1998), "Scaffolding for Success in Problem-Based Learning," *Medical Education Online*, 3(4), pp.1-15. http://www.utmb.edu/meo/

Hartley, D., Hruschka, P., and Pirbhai, I. (2000). *Process for System Architecture and Requirements Engineering*. Dorset House Publishing.

Inmon, W.H., Zachman, J.A., and Geiger, J.G. (1997). *Data Stores, Data Warehousing, and the Zachman Framework: Managing Enterprise Knowledge*. McGraw Hill.

ISO (International Standards Organization). (1996). *Reference Model for Open Distributed Processing*. International Standard 10746-1, ITU Recommendation X.901..

Jacobson, I., Christerson, M., Jonsson, P.M., and Overgaard, G. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA: Addison-Wesley.

Leffingwell, D., and Widrig, D. (2000). *Managing Software Requirements: A Unified Approach*. Addison-Wesley, Reading, Mass.

Lethbridge, T.C. (2000), "What Knowledge is Important to a Software Professional?" *IEEE Internet Computing*, May, pp. 44-50.

McConnell, S., and Tripp, L. (1999), "Professional Software Engineering: Fact or Fiction?" *IEEE Software*, Nov.-Dec., , pp. 13-18.

Meyer, B. (2001), "Software Engineering in the Academy," *IEEE Computer*, May, pp.28-35.

Parnas, D.L. (1999), "Software Engineering Programs Are Not Computer Science Programs," *IEEE Software*, Nov.-Dec., , pp.19-30.

Perkins, D.N. (1992), "What constructivism demands of the learners?" In T.M. Duffy & D.H. Jonassen (Eds.), *Constructivism and the Technology of Instruction: A Conversation* (pp. 161-165). Hillsdale, NJ: Erlbaum.

Pour, G., Griss, M.L., and Lutz, M. (2000), "The Push to Make Software Engineering Respectable," *IEEE Internet Computing*, May, pp. 35-43.

Repenning, A., Ioannidou, A, Payton, M, et al. (2001), "Using Components for Rapid Distributed Software Development," *IEEE Software*, Vol. 18, No. 2, March/April, pp. 38-45.

Ryan, G. (1993), "Student Perceptions about Self-directed Learning in a Professional Course Implementing Problem-Based Learning," *Studies in Higher Education*, Vol. 18, pp. 53-63.

Savery, J.R., and Duffy, T.M. (1995), "Problem-Based Learning: An Instructional Model and its Constructivist Framework," *Educational Technology*, 35(5), pp. 31-38.

Shaw, M. (1990), "Prospects for an Engineering Discipline of Software," *IEEE Software*, Nov., pp. 15-24.

Software Engineering (1969): Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 1968, P. Naur and B. Randell, eds., Scientific Affairs Div., NATO, Brussels.

Speed, J.R. (1999), "What Do You Mean I Can't Call Myself a Software Engineer?" *IEEE Software*, Nov.-Dec., pp.45-50.

Stapleton, J. (1997), *DSDM: Dynamic Systems Development Method – The Method in Practice*. Addison Wesley.

Umar, A. (1997), *Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies*. Prentice Hall PTR.

Vat, K.H. (2001), "Teaching HCI with Scenario-Based Design: The Constructivist's Synthesis," In Proceedings of the Sixth Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE2001), Canterbury, U.K., Jun. 25-27 , pp. 9-12.

Vat, K.H. (2000), "Teaching Software Psychology: Expanding the Perspective," In Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education, Austin, TX, Mar. 8-12, pp. 392-396.

Wasserman, A.I. (1996), "Toward a Discipline of Software Engineering," *IEEE Software*, Nov., pp. 23-31.

Wills, C.E. (1998), "Group-Based Software Engineering in an Introductory Computer Science course," In Proceedings of IEEE International Conference on Software Engineering: Education & Practice, Dunedin, New Zealand, Jan. 26-29.

Zachman, John A. (1987), "A Framework for Information Systems Architecture," *IBM Systems Journal*, Vol. 26, No. 3, IBM Publication G321-5298.

Biography

Kam Hou VAT is currently a lecturer in the Department of Computer and Information Science, under the Faculty of Science and Technology, at the University of Macau, Macau SAR, China. His current research interests include learner-centered design with constructivism in Software Engineering, architected applications developments for Internet software, information systems for learning organization, information technology for knowledge synthesis, and collaborative technologies in electronic organizations.